Package tokcycle (v1.4)

Steven B Segletes <SSegletes@verizon.net>
contributor: Christian Tellechea[1]
May 27, 2021

The tokcycle package helps one to build tools to process tokens from an input stream. If a macro to process an arbitrary single (non-macro, non-space) token can be built, then tokcycle can provide a wrapper for cycling through an input stream (including macros, spaces, and groups) on a token-by-token basis, using the provided macro on each successive character.

tokcycle characterizes each successive token in the input stream as a *Character*, a *Group*, a *Macro*, or a *Space*. Each of these token categories are processed with a unique directive, to bring about the desired effect of the token cycle. *If*-condition flags are provided to identify active, implicit, and catcode-6 tokens as they are digested. The package provides a number of options for handling groups.

Contents

1 The tokcycle macros and environments **2**
1.1 Provided (built-in) tokcycle macros and environments 3
1.2 Create your own tokcycle environments 4
1.3 Externally specified directives and directive resets 5

2 Commands in the tokcycle directives **5**
2.1 Adding tokens to the output stream: `\addcytoks` 6
2.1.1 `#1` 6
2.1.2 Transforming the input stream 6
2.2 *Group* directive: `\ifstripgrouping`, and `\processtoks` 7
2.2.1 The `\groupedcytoks`macro for more sophisticated group processing 7
2.2.2 Implicit grouping tokens: `\stripimplicitgroupingcase` 8
2.3 Looking ahead at the input stream 8
2.3.1 `\tcpeek` 9
2.3.2 `\tcpop` 9
2.3.3 `\tcpopliteral` 10
2.3.4 `\tcpopappto` and `\tcpopliteralappto` 10
2.3.5 `\tcpopuntil` 10
2.3.6 `\tcpopwhitespace` 10
2.3.7 `\tcpush` 11

[1] I am extremely grateful to Christian <unbonpetit@netc.fr> for his assistance in the development of this package. The `\addcytoks` macro was provided by him. He gave constant reminders on what the parser should be able to achieve, thus motivating me to spend the extra time striving for a generality of application that would not come naturally to me. I value highly his collegiality and hold his expertise in the highest regard.

2.3.8 \tcpushgroup 11
2.3.9 \tcappto#1from#2 11
2.4 Truncating the input stream 12
2.4.1 \truncategroup and \truncategroupiftokis 12
2.4.2 \truncatecycle and \truncatecycleiftokis 13
2.4.3 Truncating from within the *Group* Directive 13
2.5 Escaping content from tokcycle processing 13
2.6 Flagged tokens 14
2.6.1 Active characters 14
2.6.2 Implicit tokens: \ifimplicittok 14
2.6.3 Active-implicit tokens, including spaces 15
2.6.4 Parameter (cat-6) tokens (e.g., #): \ifcatSIX 15
2.6.5 Parameters (#1–#9): \whennotprocessingparameter . . 15
2.7 Misc: general *if*-condition tools 16
2.8 tokcycle completion: \aftertokcycle and \tcendgroup 16
2.9 Accommodating catcode-1,2 changes: \settcGrouping 16

3 Usage Examples **17**

4 Summary of known package limitations **17**

1 The tokcycle macros and environments

The purpose of the tokcycle package is to provide a tool to assist the user in developing token-processing macros and environments. The types of processing are limited only by the creativity of the user, but examples might include letter-case-operations, letter spacing, dimensional manipulation, simple-ciphering, {group} manipulation, macro removal, etc. In one sense, it can be thought of as a streaming editor that operates on LaTeX input streams.

The package can be loaded into both plain TeX, by way of the invocation `\input tokcycle.tex` as well as LaTeX, via `\usepackage{tokcycle}`. It provides a total of 6 macros/pseudo environments, based on three criteria:

- Two pseudo-environments with the phrase "`tokencycle`" in the name, and four macros containing the phrase "`tokcycle`". The pseudo-environments operate within a group and typeset their result upon completion. The macros operate within the document's current scope, but do not typeset the result automatically. In the case of both macros and pseudo-environments, the transformed result is available for later use, being stored in the package token list named `\cytoks`.

- Two macros and one pseudo-environment containing the phrase "`xpress`". Without the phrase, the macro/environment requires four *processing directives* to be *explicitly* specified, followed by the input stream. With the phrase present, only the input stream is to be provided. In the `xpress` case, the *processing directives* are to have been separately specified via external

macro and/or are taken from the most recent tokcycle macro invocation (failing that, are taken from the package initialization).

- Two macros containing the phrase "`expanded`". When present, the input stream of the token cycle is subject to the new TEX primitive, `\expanded`, prior to processing by the tokcycle macro. Expansion of specific macros in the input stream can be inhibited in the input stream with the use of `\noexpand`. Note that there are no `expanded` environments in tokcycle, as tokcycle environments do not pre-tokenize their input stream.

The basic approach of all tokcycle macros/environments is to grab each successive token (or group) from the input stream, decide what category it is, and use the currently active processing directives to make any desired transformation of the token (or group). Generally, with rare exception, the processed tokens should be stored in the token register `\cytoks`, using the tools provided by the package. The cycle continues until the input stream is terminated.

As tokens/groups are read from the input stream, they are categorized according to four type classifications, and are subjected to the user-specified processing directive associated with that category of token/group. The tokcycle categories by which the input stream is dissected include *Character*, *Group*, *Macro*, and *Space* (alphabetized for easy recall).

Catcode-0 tokens are directed to the *Macro* directive for processing. Catcode-10 tokens are directed to the *Space* directive. When an explicit catcode-1 token[2] is encountered in the tokcycle input stream, the contents of the entire group (sans the grouping) are directed to the *Group* directive for further processing, which may in turn, redirect the individual tokens to the other categories. The handling options of implicit cat-1 and 2 tokens are described later in this document (section 2.2.2). Valid tokens that are neither catcode 0, 1, 2, nor 10, except where noted, are directed to the *Character* directive for processing.

1.1 Provided (built-in) tokcycle macros and environments

The syntax of the non-`xpress` macros/environments is

`\tokcycle` *or* `\expandedtokcycle`
`{<`*Character processing directive*`>}`
`{<`*Group-content processing directive*`>}`
`{<`*Macro processing directive*`>}`
`{<`*Space processing directive*`>}`
`{<`*token input stream*`>}`

or, alternately, for the pseudo-environment,

`\tokencycle`
`{<`*Character processing directive*`>}`
`{<`*Group-content processing directive*`>}`
`{<`*Macro processing directive*`>}`
`{<`*Space processing directive*`>}<`*token input stream*`>\endtokencycle`

[2]Throughout this document the terms *catcode-* and *cat-* are used interchangeably.

For the `xpress` macros, the syntax is

`\tokcyclexpress` *or* `\expandedtokcyclexpress`
`{<`*token input stream*`>}`

or, alternately, for the `xpress`-pseudo-environment,

`\tokencyclexpress<`*token input stream*`>\endtokencyclexpress`

1.2 Create your own tokcycle environments

In addition to the above macros/environments, the means is provided to define new tokcycle environments:

`\tokcycleenvironment\environment_name`
`{<`*Character processing directive*`>}`
`{<`*Group-content processing directive*`>}`
`{<`*Macro processing directive*`>}`
`{<`*Space processing directive*`>}`

This will then permit simplified invocations of the form

`\environment_name<`*token input stream*`>\endenvironment_name`

More recently, an even more versatile *extended* version has been made available:

`\xtokcycleenvironment\environment_name`
`{<`*Character processing directive*`>}`
`{<`*Group-content processing directive*`>}`
`{<`*Macro processing directive*`>}`
`{<`*Space processing directive*`>}`
`{<`*Set-up code*`>}`
`{<`*Close-out code*`>}`

This will also permit simplified invocations of the form

`\environment_name<`*token input stream*`>\endenvironment_name`

However, with this extended environment definition, additional set-up and close-out code will be executed prior to and following the token cycle, within the scope of the environment. The close-out code, while executed after the token cycle, occurs before the `\cytoks` token list is actually typeset; therefore the close-out code can both execute macros and/or add tokens to the imminently-typeset token list. Any definitions or counters that change as a result of executing the *C-G-M-S* directives are, in fact, *past tense* when the close-out code is executed.

Finally, the set-up and close-out code, as well as the *C-G-M-S* directives can invoke `\tcafterenv`, whose argument is executed upon the exit from the scope of *this* defined environment.[3] As with all tokcycle environments, those created by way of `\tokcycleenvironment` and `\xtokcycleenvironment`, while operating in the scope of a group, will preserve the contents of `\cytoks` when exiting their scope.

[3]The `\tcafterenv` macro is recognized *only* in the context of an extended `\xtokcycleenvironment` definition. It is not available for direct use in the package's `\tokencycle` pseudo-environment, nor in `\tokcycleenvironment` definitions.

1.3 Externally specified directives and directive resets

For use in `xpress` mode, the directives for the *C-G-M-S* categories may be externally pre-specified, respectively, via the four macros `\Characterdirective`, `\Groupdirective`, `\Macrodirective`, and `\Spacedirective`, each taking an argument containing the particulars of the directive specification.

Each of these directives may be individually reset to the package default with the following argument-free invocations: `\resetCharacterdirective`, `\resetGroupdirective`, `\resetMacrodirective`, or `\resetSpacedirective`. In addition, `\resettokcycle` is also provided, which not only resets all four directives collectively, but it also resets, to the default configuration, the manner in which explicit and implicit group tokens are processed. Finally, it resets the `\aftertokcycle` macro to empty.

The default directives at package outset and upon reset are

```
\Characterdirective{\addcytoks{#1}}
\Groupdirective{\processtoks{#1}}
\Macrodirective{\addcytoks{#1}}
\Spacedirective{\addcytoks{#1}}
\aftertokcycle{}
\stripgroupingfalse
\stripimplicitgroupingcase{0}
```

The interpretation of these directives will be explained in the remainder of this document. Let it suffice for now to say that the default directive settings pass through the input stream to the output stream, without change.[4]

2 Commands in the tokcycle directives

The document-level token cycling tools provided in the package are listed in section 1. For each of those commands and/or pseudo-environments, the user must (explicitly or implicitly) detail a set of directives to specify the manner in which the *Character*, *Group*, *Macro*, and *Space* tokens found in the input stream are to be processed. The *C-G-M-S* processing directives consist of normal TEX/LATEX commands and user-defined macros to accomplish the desired effect. There are, however, several macros provided by the package to assist the user in this endeavor.

The recommended way to apply this package is to collect the tokcycle-transformed results of the input stream in a token register provided by the package, named `\cytoks`. Its contents can then be typeset via `\the\cytoks`.[5] The macro for appending things to `\cytoks`, to be used in the package directives, is `\addcytoks`.

[4] Except, possibly, in the case of catcode-6 tokens, which will be later addressed in sections 2.6.4 and 2.6.5.

[5] If a token-cycle input stream contains no macros, or is to be detokenized, or if the input-stream tokens are not to be typeset, it may be possible (though not required) to bypass `\cytoks` and typeset the output directly.

2.1 Adding tokens to the output stream: `\addcytoks`

The macro provided to append tokens to the `\cytoks` token register is named `\addcytoks[]{}`. Its mandatory argument consists of tokens denoting *what* you would like to append to the `\cytoks` register, while the optional argument denotes *how* you would like them appended (valid options include positive integers `[<`*n*`>]` and the letter `[x]`).

When the optional argument is omitted, the specified tokens are appended *literally* to the register (without expansion). An integer option, call it *n*, takes the the mandatory argument, and expands it *n*-times, and appends the result to `\cytoks`. The `[x]` option employs the `\expanded` primitive to maximally expand the argument before appending it to `\cytoks`.

The `[x]` option will prove useful when the *Character* or other directives involve a transformation that is fully expandable. Its use will allow the expanded result of the transformation to be placed in the token list, rather than the unexpanded transformation instructions.

2.1.1 `#1`

In the context of the *C*, *G*, *M*, and *S* processing directives, one may refer to `#1` (for example, in the argument to `\addcytoks` or `\processtoks`). TEX users know that the first parameter to a TEX macro is denoted as `#1`. The specification of all tokcycle processing directives is structured in such a way that "`#1`" may be directly employed to indicate the current token (or group) under consideration in the input stream.

2.1.2 Transforming the input stream

Within the tokcycle *C-G-M-S* directives, the command `\addcytoks{#1}` is used to echo the token being processed to the output stream that is being collected in the `\cytoks` token list. However, one may do more than merely echo the input—one may transform it. If the user creates a macro, let's call it `\xform`, to take a single character of input and convert it in some way to something else, then this macro can be used inside the *Character* directive as `\addcytoks{\xform{#1}}`. In this case, if an '`e`' were the token under consideration by the *Character* directive, '`\xform{e}`' would be added to the `\cytoks` token list. Since tokcycle operates upon each successive token in the input stream, this `\xform` macro that operates on a single token can be used to completely transform the tokcycle input stream.

If the `\xform` macro is expandable, reaching its termination in a known number, *n*, of expansions, `\addcytoks` can take advantage of that with its optional argument, performing the said number of expansions of `\xform` *before* adding the result to `\cytoks`. If the expandability of `\xform` is known, but the number *n* is not, the optional argument `[x]` to `\addcytoks` may be used to fully expand the argument (by way of the `\expanded` primitive). In that case, the resulting `\cytoks` result will show no evidence of the `\xform` macro, but only the results of its transformation.

The accompanying examples document, `tokcycle-examples.pdf`, is full of examples showing specific tokcycle-induced transformations.

2.2 *Group* directive: `\ifstripgrouping`, and `\processtoks`

The *Group* directive is unique, in that it is the only directive whose argument (`#1`) may consist of more than a single token. There are two issues to consider when handling the tokens comprising a group: do I retain or remove the grouping (cat-1,2 tokens)? Do I process the group's token content individually through the token cycle, or collectively as a unit?

Grouping in the output stream is determined by the externally set condition `\ifstripgrouping`. The package default is `\stripgroupingfalse`, such that any explicit grouping that appears in the input stream will be echoed in the output stream. The alternative, `\stripgroupingtrue`, is dangerous in the sense that it will strip the cat-1,2 grouping from the group's tokens, thereby affecting or even breaking code that utilizes such grouping for macro arguments. Apply `\stripgroupingtrue` with care.

The issue of treating the tokens comprising the content of a group individually or collectively is determined by the choice of macro inside the *Group* directive. Within the *Group* directive, the argument `#1` contains the collective tokens of the group (absent the outer cat-1,2 grouping). The default directive `\processtoks{#1}` will recommit the group's tokens individually to be processed through the token cycle. In contrast, the command `\addcytoks{#1}` in the *Group* directive would directly add the collective group of tokens to the `\cytoks` register, without being processed individually by tokcycle.

2.2.1 The `\groupedcytoks` macro for more sophisticated group processing

To this point, there have been presented two options for handling grouped material in the token cycle: process each of the grouped tokens (via `\processtoks`) or echo the grouped tokens (via `\addcytoks`). Obviously, one may also discard the grouped tokens altogether by employing neither of the above choices in the context of the *Group* directive.

However, there are times where you might wish to perform a macro task (for example, one that might add additional tokens to `\cytoks`) immediately *outside* of the group, before or after. If you were to add the task macro before or after the `\processtoks` invocation (anywhere in the *Group* directive), it will still be performed, by default, *inside* the `\cytoks` group.

The macro `\groupedcytoks` allows one to manually specify the grouping duties of the *Group* directive. Thus, the following two group-processing configurations are functionally identical: the default

```
\stripgroupingfalse
\Groupdirective{\processtoks{#1}}
```

and

```
\stripgroupingtrue
\Groupdirective{\groupedcytoks{\processtoks{#1}}}
```

Therefore, unless the goal is to introduce additional nesting levels into the input stream, the first rule of using `\groupedcytoks` is to `\stripgroupingtrue` before entering the token cycle. The argument of `\groupedcytoks` specifies the tasks that are to occur inside a `\cytoks` grouping. So, in the following example

```
\stripgroupingtrue
\Groupdirective{\taskA
  \groupedcytoks{\taskB\processtoks{#1}\taskC}%
  \taskD}
```

the tasks A and D will occur *outside* of the `\cytoks` group. Tasks B and C occur inside the group, immediately before and after processing the tokens of the group, respectively. Thus, the use of `\groupedcytoks` in this way permits tokcycle taskings to occur outside of the grouping applied to the output.

If `\stripgroupingtrue` had been omitted while still using `\groupedcytoks`, tasks A and D would have been in the explicit group, and tasks B and C, along with the grouped tokens `#1`, would have been nested within an additional group.

2.2.2 Implicit grouping tokens: `\stripimplicitgroupingcase`

Implicit grouping tokens (e.g., `\bgroup` & `\egroup`) can be handled in one of *three* separate ways. Therefore, rather than using an *if*-condition, the external declaration `\stripimplicitgroupingcase{}` is provided, which takes one of 3 integers as its argument (0, 1, or −1). The package-default case of "0" indicates that the implicit-group tokens will not be stripped, but rather echoed directly into the output stream. The case of "1" indicates that the implicit-group tokens will be stripped and not placed into the output stream (as with explicit grouping, this is a dangerous case and should be specified with care).

Finally, the special case of −1 indicates that the implicit-group tokens should instead be passed to the *Character* directive for further handling (note that the `\implicittoktrue` condition will be set[6]). Such a special treatment has limited application—for example, when the intent is to detokenize these tokens.

2.3 Looking ahead at the input stream

In the normal mode of tokcycle operation, each token of the input stream is successively digested and sent to one of four user-defined directives for processing. Such an approach works well if there is no interdependency of adjacent tokens in the input stream. When such an interdepency exists (for example, the argument associated with an invoked macro), one might typically use flags that are set or cleared in one directive that can then be status-checked when the subsequent token is processed. While such an approach is wholly valid, it can create a complex web of flag setting/checking that can span across multiple directives.

[6] as well as the internal condition `\tc@implicitgrptrue`

With the release of v1.4 of the package, an alternative approach to handle such dependencies has been developed: look-ahead features, so that the occurrence of a particular token or condition can provoke an immediate examination of the input stream to handle a possible dependency, without waiting for the next iteration of the token cycle.

It should be noted that the use of look-ahead is perhaps more *dangerous* than the traditional use of flags—for if it is not performed with care, tampering with the input stream can destroy its integrity. The commands developed for these situations will now be described. They are designed to be employed as part of tokcycle's *Character*, *Macro*, and *Space* directives for checking and/or handling successively linked tokens in the input stream. They cannot be used inside the *Group* directive.

2.3.1 `\tcpeek`

When issued witin a directive as, for example `\tcpeek\z`, the next token from the input stream is `\futurelet` into `\z`. This has several implications. The input stream remains undisturbed, so that the peeked-at token will still be the token to be digested in the next iteration of the token cycle. Because it has been `\let`, `\z` does not *contain* the token from the input stream in the manner of a `\def`, but rather it is a separate token *possessing the same properties* as the input-stream token. As such, it is ideal to be used for `\ifx` comparisons. For example, `\ifx\bgroup\z` will detect whether a cat-1 group-opening token is at the head of the input stream, e.g., an opening brace, '{'; the comparison `\ifx\tcsptoken\z` will detect whether a cat-10 space is at the head of the input stream. Such comparisons can be useful in directing the logic of the tokcycle directive.

2.3.2 `\tcpop`

When issued as, for example `\tcpop\z`, the next token(s) from the input stream is *immediately* digested as an argument and the tokens are placed into the macro `\z`. This token (or tokens) will no longer be digested as part of the next iteration within the token cycle, unless the tokens are subsequently replaced at the head of the input stream (see `\tcpush` below).

Beware that when TEX absorbs an argument, leading white space is lost. Further, if the argument was enclosed in cat-1,2 braces, the braces are stripped and the whole group is absorbed as the argument. In some cases, if the argument needs manipulation, this brace-stripping behavior may be desired. However, if the retention of the leading white space and the braces are desired, one can use manual techniques (possibly involving macros such as `\tcpeek`, `\tcpopwhitespace` and `\tcpushgroup`) or one can instead use `\tcpopliteral` (see below) in lieu of `\tcpop`.

If `\tcpop` is used nonetheless, to aid in such matters when cat-10 space is at the head of the input stream, the flag `\spacepoppedtrue` will be set when a `\tcpop` is issued. However, even `\ifspacepopped` will be unable to differentiate

explicit "white" space that is lost during argument absorption versus implicit space (e.g., `\@sptoken` or active-implicit space) that is not discarded. Another option for dealing with leading white space is `\tcpopwhitespace` (see below).

2.3.3 `\tcpopliteral`

This macro is an alternative to `\tcpop`, if the retention of leading white space and possibly braces are required when popping an argument from the input stream. This can be particularly useful if the popped tokens must later be placed back into the input stream in their original state. The syntax, `\tcpopliteral\z`, pops an argument into `\z`, while retaining possible leading whitespace and brace groups.

2.3.4 `\tcpopappto` and `\tcpopliteralappto`

When issued as, for example `\tcpopappto\z`, the next token(s) from the input stream is absorbed as an argument and *appended* to the replacement text of `\z`. The same group/space provisos affecting `\tcpop` apply here as well. The control sequence to be appended, here `\z`, may not be undefined when `\tcpopappto` is invoked. This macro is just a convenient joining of two macros: `\tcpop` and `\tcappto#1from#2` (see below).

The same applies for the macro `\tcpopliteralappto`, which conveniently joins `\tcpopliteral` and `\tcappto#1from#2`.

2.3.5 `\tcpopuntil`

This command pops one or more tokens from the input stream in the manner of a delimited argument. Thus, when issued as, for example `\tcpopuntil 0\z`, tokens from the input stream are absorbed into `\z` until an `0` token is reached. Unlike delimited-argument absorption, however, the delimiter (`0` in this example) is also added to the specified control sequence, in this case `\z`. This construct is very useful for obtaining optional-argument tokens from the input stream. Consider the following code, with an input stream of `[1ex]{2ex}`:

```
\tcpeek\z
\ifx[\z\tcpopuntil ]\q\else\def\q{}\fi
```

Since an optional argument is next in the input stream, `\q` will obtain the replacement text `[1ex]`, and the input stream will now begin with `{2ex}`. If an optional argument had not been next in the input stream, then `\q` would be empty and the input stream would still begin with `{2ex}`.

2.3.6 `\tcpopwhitespace`

When issued as, for example `\tcpopwhitespace\z`, leading white space from the input stream will be absorbed and set in `\z`, in the manner of a `\def`. Unlike the flag `\ifspacepopped`, leading implicit space will not be indicated by this macro—only leading explicit white space. If no leading white space is present,

\z will be empty following its invocation and no tokens will have been removed from the input stream.

2.3.7 \tcpush

To this point, several commands have been described for reading and/or extracting tokens from the input stream. There are also a corresponding commands for placing material at the head of the input stream, should that need arise. With an invocation of \tcpush\z, the replacement text of \z will be pushed onto the input stream. Multiple pushes will be handled in a last-in-first-out fashion.

Beware that, because argument absorption in TEX will strip the braces of an absorbed group, care must be taken. If the input stream leads with a grouped quantity, such as {abc}, then \tcpop\z\tcpush\z will end up with abc in the input stream, with the braces missing. *The commands \tcpop and \tcpush are not strictly inverse operations* (see \tcpopliteral as an alternative). For this reason, the command \tcpushgroup is also provided (see below).

The command \tcpush supports an optional argument that functions in the same manner as that of \addcytoks, to provide additional levels of expansion beyond the mere replacement text. Thus, \tcpush[2]\z will take the replacement text of \z, expand it twice, and push the result onto the input stream. In a similar way, \tcpush[x]\z will fully expand the replacement text of \z and then place those tokens onto the input stream. Whereas \addytoks places tokens into the output stream (the token list \cytoks), \tcpush places tokens at the head of the input stream.

2.3.8 \tcpushgroup

The command \tcpushgroup functions in a manner similar to \tcpush, except that the replacement text of the argument is placed onto the input stream within a braced (cat-1,2) group. So if the replacement text of \z is abc, then the invocation \tcpushgroup\z will place {abc} onto the input stream. As with \tcpush, and with the identical syntax, the command \tcpushgroup supports an optional argument that directs addition levels of expansion beyond the mere replacement text.

2.3.9 \tcappto#1from#2

This macro does not touch the input stream, per se. However, as tokens are popped from the input stream and placed into macros, this command conveniently allows for their aggregation Thus, if \q contains abc and the newly popped \z contains d, then \tcappto \q from \z will append the contents of \z to \q, so that, upon conclusion, \q will contain the tokens abcd. Unexpandable tokens can also be appended directly using this macro, using the syntax of \tcappto \q from{123}. Expandable tokens can also be added directly with the use of this macro, by using a leading \noexpand or \empty, as in \tcappto \q from{\noexpand\today}.

2.4 Truncating the input stream

The basic process of tokcycle is one in which an input stream (or argument) of tokens is examined and processed based on directives set up by the user. Typically, the input stream is processed token-by-token to exhaustion, which occurs when a defined terminating token is reached. In section 2.5, we will see how a sequence of tokens can be exempted from any processing that would otherwise occur in the directive, and be passed instead directly to the output stream.

However, in this section, we will examine how tokcycle can be directed to *dynamically* truncate, that is, discard the remaining input stream based on the tokens found therein. Truncation commands may be issued within the *Character*, *Macro*, and *Space* directives. More will be said about the *Group* directive in section 2.4.3.

Truncation can be made to apply either to the remainder of the current tokcycle-nesting level (corresponding to an explicit catcode-1,2 group in the input stream)[7] or for the remainder of the total tokcycle input stream.

2.4.1 `\truncategroup` and `\truncategroupiftokis`

As to truncating the input stream for the remainder of the tokcycle nesting level, the command is simply `\truncategroup`. So, for example, the directive

```
\Spacedirective{\truncategroup}
```

would direct tokcycle, if it ever finds a (catcode-10) space token in the input stream, to discard the remainder of the tokens within the group in which that space was initially found.

More often, however, one would desire to issue the truncation *conditionally*. One could use one of the conditional commands of section 2.7, for example,

`\tctestifcon{<`*condition*`>}{\truncategroup}{<`*code if condition not met*`>}`

Alternatively, if the *condition* is the occurrence of a particular single token in the input stream, one may use an abbreviated syntax:

`\truncategroupiftokis{<`*tok*`>}{<`*code if condition not met*`>}`

Thus, an example to echo macro tokens to the output stream, unless a `\relax` token is found, in which case terminate the group, would be

```
\Macrodirective{\truncategroupiftokis{\relax}{\addcytoks{#1}}}
```

Obviously, checks for particular character tokens would be placed in the *Character* directive, rather than the *Macro* directive.

[7] Note that implicit groups (`\bgroup...\egroup`) do not create a new tokcycle nesting level, nor do `\begingroup...\endgroup`. Only explicit catcode-1,2 groups, in the nature of `{...}`, create a nested tokcycle level. The current tokcycle nesting level may be obtained from the TEX count `\tcdepth`, which starts at zero upon tokcycle entry. This count is incremented and decremented as groups are entered and exited in the input stream. It may be examined within the *C-G-M-S* directives in order to guide decisions.

2.4.2 `\truncatecycle` and `\truncatecycleiftokis`

There are two commands that are in every way analogous to the group-terminating macros of section 2.4.1. These commands are `\truncatecycle` and `\truncatecycleiftokis`. But in this case, all tokens are discarded to the end of the input stream, not just the current group (i.e., tokcycle-nesting level). One point to note, however, is that open group levels will be closed by the `\truncatecycle` command, so that the tokens, both those being executed as well as those retained in `\cytoks`, will be group-balanced.

2.4.3 Truncating from within the *Group* Directive

The truncate commands described in prior sections may *not* be executed from the *Group* directive, as has already been mentioned. What does termination even mean in the `\Groupdirective` context? The *Group* directive is executed whenever an explicit catcode-1,2 group is encountered in the input stream. The argument, `#1`, to that directive will be the complete contents of the group (sans the catcode-1,2 delimiters).

So, to truncate the group from the outset, not doing anything with `#1` will accomplish the desired result. For example, to truncate all group-level-3 tokens, you could say, using the conditional commands of section 2.7,

```
\Groupdirective{\tctestifnum{\tcdepth=3}{}{\processtoks{#1}}}
```

There is no need to use an explicit `\truncategroup` macro to accomplish it.

On the other hand, to truncate the complete token cycle if, for example, group-level-3 tokens are encountered, you have to set some sort of flag that is immediately picked up by one of the other directives. Here is an example of how that can be done:

```
\Groupdirective{\tctestifnum{\tcdepth=3}{\processtoks{\truncatenow}}%
                                        {\processtoks{#1}}}
\Macrodirective{\truncatecycleiftokis{\truncatenow}{\addcytoks{#1}}}
```

Note that, as long as `\stripgroupingfalse` is active, the above examples will result in empty, rather than absent, level-3 groups, since the group open occurs before the truncation is processed. If the presence of even an empty group is to be avoided in such a case, one may employ the `\groupedcytoks` techniques described in section 2.2.1.

2.5 Escaping content from tokcycle processing

There are times you may wish to prevent tokens in the tokcycle input stream from being operated on by tokcycle. Rather, you just want the content passed through unchanged to the output; that is, with the intent to have multi-token content bypass the tokcycle directives altogether.

The method developed by the package is to enclose the escaped content in the input stream between a set of tokcycle escape characters, initially set to a vertical rule character found on the keyboard: "|". The main proviso

is that the escaped content cannot straddle a group boundary (the complete group may, however, be escaped). The escape character can be changed with `\settcEscapechar{<`*escape-token*`>}`.

2.6 Flagged tokens

Certain token types are trapped and flagged via true/false *if*-conditions. These *if*-conditions can be examined within the appropriate directive (generally the *Character* directive), to direct the course of action within the directive.

2.6.1 Active characters

`\ifactivetok`: Active (cat-13) tokens that occur in the input stream result in the flag `\ifactivetok` being set `\activetoktrue`. Note that the expansion of the token's active `\def` occurs *after* tokcycle processing. With active `\let`'s, there is no text substitution; however, the assignment is already active at the time of tokcycle processing. The only exception to this rule is with pre-expanded input, `\expandedtokcycle[xpress]`. If an active token's substitution is governed by a `\def`, the text substitution will have occurred before reaching the token cycle.

`\ifactivetokunexpandable`: This flag is similar to `\ifactivetok`, in that a token must be active for this to be set true. However, in addition, it is only true if the active token is `\let` to a character or a primitive, neither of which can be expanded. Active characters assigned via `\def` or else `\let` to a macro will *not* qualify as `\activetokunexpandabletrue`.

`\ifactivechar`: This flag, rather than testing the token, tests the character code of the token, to see if it is set active. Generally, the token and its character code will be syncronized in their *activeness*. However, if a token is tokenized when active, but the corresponding character code is made non-active in the meantime, prior to the token reaching tokcycle processing, this flag will be set `\activecharfalse`. A similar discrepency will arise if a token is not active when tokenized, but the character code is made active in the interim, prior to tokcycle processing.

2.6.2 Implicit tokens: `\ifimplicittok`

Implicit tokens, those assigned via `\let` to characters[8], are flagged with a `true` setting for `\ifimplicittok`. Generally, implicit tokens will be directed to the

[8]Some clarification may be needed. Control sequences and active characters that are `\let` to something other than a cat-0 control sequence will be flagged as implicit. If implicit, a token will be processed through the *Character* directive (exceptions noted). On the other hand, if a control sequence or active character is `\let` to a cat-0 control sequence, it will be directed to the *Macro* directive for processing, without the implicit flag.

Character directive for processing. There are, however, two exceptions: i) implicit grouping tokens (e.g., `\bgroup` and `\egroup`) will not appear in any directive unless the `\stripimplicitgroupingcase` has been set to −1; and ii) implicit space tokens (e.g., `\@sptoken`) will be processed by the *Space* directive.

2.6.3 Active-implicit tokens, including spaces

One may occasionally run across a token that is both active and implicit. For example, in the following code, `Q` is made both active and implicit:

```
\catcode`Q=\active
\let Qx
```

In general, both `\ifactivetok` and `\ifimplicittok` tests can be performed together to determine such cases.

This is true even in the case of active-implicit catcode-10 spaces, which are always processed through the *Space* directive. As of tokcycle v1.4, the actual active-implicit space, if encountered in the input stream, will be passed as `#1` to the *Space* directive, *as long as the character code of the token is still* `\active`. If the character code of that token is no longer active, a generic implicit space token named `\tcsptoken` is instead passed to the the *Space* directive as `#1`. In either case, the catcode-12 version of the active character that was digested will be, for that moment, retained in a definition named `\theactivespace`. This can be useful if detokenization is required of the spaces. Such an example is described in the `tokcycle-examples` adjunct document.

2.6.4 Parameter (cat-6) tokens (e.g., #): `\ifcatSIX`

Typically, category-code 6 tokens (like `#`) are used to designate a parameter (e.g., `#1`–`#9`). Since they are unlikely to be used in that capacity inside a tokcycle input stream, the package behavior is to convert them into something cat-12 and set the *if*-condition `\catSIXtrue`. In this manner, `\ifcatSIX` can be used inside the *Character* directive to trap and convert cat-6 tokens into something of the user's choosing.

As to the default nature of this conversion (if no special actions are taken), explicit cat-6 characters are converted into the identical character with category code of 12. On the other hand, implicit cat-6 macros (e.g., `\let\myhash#`) are converted into a fixed-name macro, `\implicitsixtok`, whose cat-12 substitution text is a `\string` of the original implicit-macro name.

2.6.5 Parameters (`#1`–`#9`): `\whennotprocessingparameter`

While, generally, one would not intend to use parameters in the tokcycle input stream, the package provides, not a flag, but a macro to allow it. The macro is to be used *within* the *Character* directive with the syntax:

`\whennotprocessingparameter#1{<`*non-parameter-processing-directive*`>}`

Here, the `#1` doesn't refer to `#1` as it might appear in the input stream, but to the sole parameter of the *Character* directive, referring to the current token

being processed. With this syntax, when the token under consideration is a parameter (e.g., `#1`–`#9`), that parameter is added to the `\cytoks` register. If the token under consideration is not a parameter, the code in the final argument is executed.

2.7 Misc: general *if*-condition tools

TEX comes equipped with a variety of `\if...` condition primitives. When dealing with macros for which the order of expansion is important, the `\else` and `\fi` can sometimes get in the way of proper expansion and execution. These four restructured *if* macros came in handy writing the package, and may be of use in creating your directives, preventing `\else` and `\fi` from getting in the way:

`\tctestifcon{<`*TeX-if-condition*`>}{<`*true-code*`>}{<`*false-code*`>}`
`\tctestifx{<\`*ifx-comparison-toks*`>}{<`*true-code*`>}{<`*false-code*`>}`
`\tctestifnum{<\`*ifnum-condition*`>}{<`*true-code*`>}{<`*false-code*`>}`
`\tctestifcatnx<`*tok1*`><`*tok2*`>{<`*true-code*`>}{<`*false-code*`>}`

Note that unlike those macros requiring three arguments, the lone exception is `\tctestifcatnx`, which requires four, comparing the catcodes of the two input toks **without expansion** (`nx` denotes `\noexpand`).

2.8 tokcycle completion: `\aftertokcycle` and `\tcendgroup`

The tokcycle *macros*, upon completion, do nothing. Unlike `\tokencycle` *environments*, they don't even output `\the\cytoks` token register. A command has been provided, `\aftertokcycle`, which takes an argument to denote the actions to be executed following completion of *all* subsequent tokcycle *macro* invocations within the scope of the current group. It might be specified as simply as `\aftertokcycle{\the\cytoks}`, so as to have the output stream automatically typeset by future invocations of the `\tokcycle` macro.

The meaning of `\aftertokcycle` can be reset with `\aftertokcycle{}`, but is also reset as part of `\resettokcycle`. Unlike macros, the tokcycle environments are unaffected by `\aftertokcycle`, as they actually set it locally (within their grouped scope) to accomplish their own purposes.

Speaking of the tokcycle environments, there is a command employed by the environments `\tokencycle` and `\tokencyclexpress` that is likewise available to the user: `\tcendgroup`. This is a version of `\endgroup` that additionally transmits the current value of the `\cytoks` token list *outside of the group*. So if a user wishes to confine certain tokcycle activity inside a group, but wishes to have access to the `\cytoks` result afterward, he may simply wrap the activity in `\begingroup...\tcendgroup`.

2.9 Accommodating catcode-1,2 changes: `\settcGrouping`

In order to avoid making the tokcycle parser overly complex, requiring multiple passes of the input stream, the package defaults to using catcode-1,2 braces

`{ }` to bring about grouping in the output stream, regardless of what the actual cat-1,2 tokens are in the input stream. As long as their sole purpose in the token cycle is for grouping and scoping, this arrangement will produce the expected output.

However, if the actual character-code of these tokens is important to the result (e.g., when detokenized), there is one other option. The package allows the external specification of which cat-1,2 tokens should be used in the tokcycle output stream. The syntax is `\settcGrouping{{#1}}`, to use the standard braces for this purpose (default). If angle-brackets `< >` were to be the *new* grouping tokens, then, after their catcodes were changed, and `\bgroup` and `\egroup` were reassigned, one would invoke `\settcGrouping<<#1>>`. These will then be the grouping tokens in the tokcycle output stream until set to something else.

3 Usage Examples

See the adjunct file, `tokcycle-examples.pdf`, for an array of tokcycle examples.

4 Summary of known package limitations

The goal of this package is not to build the perfect token-stream parser. It is, rather, to provide the means for users to build useful token-processing tools for their TEX/LATEX documents.

What follows are the known limitations of the package. which arise, in part, from the single-pass parsing algorithm embedded in the package. Surely, there are more cases associated with arcane catcode-changing syntax that are not accounted for; I encourage you to bring them to my attention. If I can't fix them, I can at least disclaim and declaim them below:

- One must inform the package (via `\settcGrouping`) of changes to the cat-1,2 tokens *if* there is a need to detokenize the output with the specified bracing group; however, grouping will still be handled properly (i.e., cat-1,2 tokens will be detected), even if the package is not notified. See section 2.9.

- Should one need to keep track of the *names* of implicit cat-6 tokens (e.g., `\svhash` following `\let\svhash#`), care must be exercised. When encountered in the input stream, a cat-12 `\string` of the implicit-cat-6 token *name* is stored in the `\implicitsixtok` macro. It is the `\implicitsixtok` token which is passed to the *Character* directive as `#1`, in lieu of the actual cat-6 token. If there is at most a single implicit-cat-6 named token in the input stream, then `\implicitsixtok` will reflect its name. However, if there is more than one, `\implicitsixtok` will be redefined on the fly with each occurrence.

  Therefore, should one intend to add more than one such token *name* to the token list by way of `\addcytoks`, *it must be expanded first* (e.g.,

`\addcytoks[1]{#1}`), in order to capture the *name* of that particular cat-6 implicit token, rather than the name of the last one in the input stream. There is no similar problem for explicit cat-6 tokens. In any event, all cat-6 tokens are trapped and flag as `true` the `\ifcatSIX` condition, which is how the user should detect their presence.

- A similar warning is to be made for active-implicit spaces as was made for implicit-cat-6 tokens. In the case of active-implicit spaces, the name (a cat-12 string) of the active character is momentarily stored in the `\theactivespace` macro. If one wishes to output the active *name* to the `\cytoks` token list, one must make sure that `\theactivespace` is expanded once before being added to the token list.

  One difference here, however, is that, unlike a cat-6 token, an active-implicit space (and not a string of it) is actually passed to the *Space* directive as `#1`. In all but one special case, the active-implicit `#1` will match that which occurred in the input stream (see section 2.6.3).

Acknowledgments

In addition to Christian Tellechea, a contributor to this package, the author would like to thank Dr. David Carlisle for his assistance in understanding some of the nuances of token registers. Likewise, his explanation about how a space token is defined in TEX (see `https://tex.stackexchange.com/questions/64197/pgfparser-module-and-blank-spaces/64200#64200`) proved to be useful here. The `tex.stackexchange` site provides a wonderful opportunity to interact with the leading developers and practitioners of TEX and LATEX.

Source Code

tokcycle.sty

```
\input tokcycle.tex
\ProvidesPackage\tcname[\tcdate\space V\tcver\space Cycle through and transform
  a stream of tokens]
\endinput
```

tokcycle.tex

```
\def\tcname                     {tokcycle}
\def\tcver                        {1.4}
%
\def\tcdate                    {2021/05/27}
%
% Author     : Steven B Segletes, Christian Tellechea (contributor)
% Maintainer : Steven B Segletes
% License    : Released under the LaTeX Project Public License v1.3c
%              or later, see http://www.latex-project.org/lppl.txt
% Files      : 1) tokcycle.tex
%              2) tokcycle.sty
%              3) tokcycle-doc.tex
%              4) tokcycle-doc.pdf
%              5) tokcycle-examples.tex
%              6) tokcycle-examples.pdf
%              7) README
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MACRO FORM
\long\def\tokcycle#1#2#3#4#5{\tokcycraw{#1}{#2}{#3}{#4}#5\endtokcycraw}
% \expanded-ARGUMENT MACRO FORM
\long\def\expandedtokcycle#1#2#3#4#5{\cytoks{\tokcycraw{#1}{#2}{#3}{#4}}%
  \expandafter\the\expandafter\cytoks\expanded{#5}\endtokcycraw}
% ENVIRONMENT FORM
\long\def\tokencycle#1#2#3#4{\begingroup\let\endtokencycle\endtokcycraw
  \aftertokcycle{\the\cytoks\tcendgroup}\tokcycraw{#1}{#2}{#3}{#4}}
% XPRESS-INTERFACE MACRO FORM
\long\def\tokcyclexpress#1{\tokcycrawxpress#1\endtokcycraw}
% XPRESS-INTERFACE \expanded-ARGUMENT MACRO FORM
\long\def\expandedtokcyclexpress#1{%
  \expandafter\tokcycrawxpress\expanded{#1}\endtokcycraw}
% XPRESS-INTERFACE ENVIRONMENT FORM
\def\tokencyclexpress{\begingroup\let\endtokencyclexpress\endtokcycraw
  \aftertokcycle{\the\cytoks\tcendgroup}\tokcycrawxpress}
% INITIALIZATION & INTERNAL TOOLS
\def\tcendgroup{\expandafter\endgroup\expandafter\tcenvscope\expandafter{\the\cytoks}}
\def\tcenvscope{\cytoks}% CAN SET TO \global\cytoks TO OVERCOME SCOPE LIMITS
\edef\restorecatcode{\catcode\number`\@=\number\catcode`\@\relax}
\catcode`\@11
\newif\iftc@implicitgrp
\newif\if@argnext
\newtoks\tc@tok
\newcount\tcdepth
\def\tc@gobble#1{}
\def\tc@deftok#1#2{\let#1= #2\empty}
\tc@deftok\tc@sptoken{ }
\expandafter\def\expandafter\tc@absorbSpace\space{}
\def\tc@ifempty#1{\tc@testxifx{\expandafter\relax\detokenize{#1}\relax}}
\def\tc@defx#1{\tc@earg{\def\expandafter#1}}
\long\def\tc@earg#1{\expandafter#1\expandafter}
\long\def\tc@xarg#1#2{\tc@earg#1{\expanded{#2}}}
\long\def\tc@exfirst#1#2{#1}
\long\def\tc@exsecond#1#2{#2}
\long\def\tc@testxifx{\tc@earg\tctestifx}
\long\def\test@ifmacro#1{\tctestifcatnx#1\relax}
\def\tc@addtoks#1#2{\toks#1\expandafter{\the\toks#1 #2}}
\def\add@tcdepth{\advance\tcdepth 1\relax}
```

```
\def\sub@tcdepth{\advance\tcdepth-1\relax}
\def\tc@resetifs{\activetokfalse\implicittokfalse\tc@implicitgrpfalse\catSIXfalse
  \activecharfalse\activetokunexpandablefalse\def\theactivespace{}}
\long\def\count@stringtoks#1{\tc@earg\count@toks{\string#1}}
\long\def\count@toks#1{\the\numexpr-1\count@@toks#1.\tc@endcnt}
\long\def\count@@toks#1#2\tc@endcnt{+1\tc@ifempty{#2}{\relax}{\count@@toks#2\tc@endcnt}}
\def\sv@hash{##}
\def\def@popname{\expandafter\def\tc@popname}
% EXTERNAL TOOLS
\let\tcsptoken= \tc@sptoken
\long\def\tctestifcon#1{#1\expandafter\tc@exfirst\else\expandafter\tc@exsecond\fi}
\long\def\tctestifcatnx#1#2{\tctestifcon{\ifcat\noexpand#1\noexpand#2}}
\long\def\tctestifx#1{\tctestifcon{\ifx#1}}
\long\def\tctestifnum#1{\tctestifcon{\ifnum#1\relax}}
\newif\ifstripgrouping
\def\stripimplicitgroupingcase#1{\edef\@implicitgroupingcase{\the\numexpr1-#1}}
\newif\ifcatSIX
\newif\ifimplicittok
\newif\ifactivetok
\newif\ifactivechar
\newif\ifactivetokunexpandable
\newif\ifspacepopped
\newtoks\cytoks
\long\def\tokcycleenvironment#1#2#3#4#5{\expandafter\def\expandafter#1%
  \expandafter{\expandafter\let\csname end\expandafter\tc@gobble
  \string#1\endcsname\endtokcycraw\tokencycle{#2}{#3}{#4}{#5}}}
\long\def\xtokcycleenvironment#1#2#3#4#5#6#7{\expandafter\def\expandafter
  #1\expandafter{\expandafter\let\csname end\expandafter\tc@gobble
  \string#1\endcsname\endtokcycraw\begingroup
  \long\def\tcafterenv####1{\tc@defx\tcendgroup{\tcendgroup####1}}%
  \aftertokcycle{#7\the\cytoks\tcendgroup}#6\tokcycraw{#2}{#3}{#4}{#5}}}
\long\def\processtoks#1{\@tokcycle#1\endtokcycraw}
\def\whennotprocessingparameter#1#2{\tctestifcon\if@argnext{\@argnextfalse\cytoks
  \expandafter{\the\cytoks###1}}{\tctestifcon\ifcatSIX{\@argnexttrue}{#2}}}
\long\def\truncategroup#1{\tctestifx{\endtokcycraw#1}{#1}{\truncategroup}}
\long\def\truncatecycle#1{\tctestifx{\endtokcycraw#1}{\tctestifnum{\tcdepth=0}
  {#1}{\truncatecycle}}{\tctestifx{\sub@tcdepth#1}{#1}
  {\tctestifx{\exit@grouped#1}{#1}{}}\truncatecycle}}
\long\def\truncategroupiftokis#1#2{\tctestifx{\tc@next#1}{\truncategroup}{#2}}
\long\def\truncatecycleiftokis#1#2{\tctestifx{\tc@next#1}{\truncatecycle}{#2}}
% ESSENTIAL METHOD: STREAMING MACRO WITH TERMINATOR:
% \tokcycraw{<Char>}{<Group>}{<Macro>}{<Space>}<input-stream>\endtokcycraw
\long\def\tokcycraw#1#2#3#4{\def\@chrT##1{#1}\long\def\@grpT##1{#2}%
  \long\def\@macT##1{#3}\def\@spcT##1{#4}\tokcycrawxpress}
% ENTRY POINT FOR XPRESS METHOD WITHOUT EXPLICIT ARGUMENTS
\def\tokcycrawxpress{\cytoks{}\tcdepth=0\relax\@tokcycle}
% CODE TO EXECUTE AT COMPLETION
\long\def\aftertokcycle#1{\def\@aftertokcycle{#1}}
\def\endtokcycraw{\tctestifnum{\tcdepth=0}{\@aftertokcycle}{}}
% LOOP ENTRY POINT
\def\@tokcycle{\tc@resetifs\futurelet\tc@next\detect@CANTabsorb}
\def\detect@CANTabsorb{\tctestifx{\tc@next\tc@sptoken}{\stringify\@@@@@@spcT}%
  {\tctestifx{\tc@next\bgroup}{\stringify\@@@@grpT}{\can@absorb}}}
% NON cat1,10 TOKENS
\long\def\can@absorb#1{\tc@tok{#1}\trapcatSIX{#1}\expandafter\can@absorb@
  \the\tc@tok}
\long\def\can@absorb@#1{\tctestifnum{\count@stringtoks{#1}>1}%
```

```
  {\tctestifx{\endtokcycraw#1}{#1}{\backslashcmds#1\@tokcycle}}%
  {\trapactives#1\tc@trapescape#1{\tc@escapecytoks}{\can@absorb@@#1}}}
\long\def\can@absorb@@#1{\let\@tmp=#1\test@ifmacro\@tmp{\implicittokfalse
  \@macT#1}{\trapimplicitegrp#1\implicitgrpfork#1}\@tokcycle}
%CONVERT NEXT (SPACE OR BEGIN-GROUP) TOKEN TO STRING
\def\stringify#1{\expandafter#1\string}% #1 WILL BE \@@@@@@spcT or \@@@@grpT
%SPACE DECODE
\def\@@@@@@spcT{\futurelet\tc@str\@@@@@spcT}
\def\@@@@@spcT{%
 \tctestifx{\tc@str\tc@sptoken}%
 {\def\@tmp{\@@spcT{ }}\expandafter\@tmp\tc@absorbSpace}% EXPLICIT SPACE
 {\implicittoktrue\tctestifcon{\if\expandafter\@firstoftwo\string\\\tc@str}%
   {\expandafter\@@@spcT\tc@gobble}% IMPLICIT MACRO SPACE
   {\activetoktrue\@@@@spcT}}}% IMPLICIT ACTIVE SPACE
\def\@@@@spcT#1{\def\theactivespace{#1}\trapactivechar{#1}\ifactivechar\edef\tc@spc
  {\scantokens{#1\noexpand}}\else\def\tc@spc{\tcsptoken}\fi\expandafter\@@spcT
  \expandafter{\tc@spc}}
\def\@@@spcT{\csmk{\expandafter\@@spcT\thecs}}
\def\@@spcT#1{\@spcT{#1}\@tokcycle}
% GROUP DECODE
\def\@@@@grpT{\futurelet\tc@str\@@@grpT}
\def\@@@grpT#1{\tctestifnum{\number\catcode`#1=1}%
  {\expandafter\@@grpT\expandafter{\iffalse}\fi}% {
  {\implicittoktrue\tc@implicitgrptrue%
    \tctestifnum{`#1=92}% WORKS EVEN IF CAT-0 HAS CHANGED
    {\csmk{\expandafter\backslashcmds\thecs\@tokcycle}}% \bgroup
    {\begingroup\catcode`#1=\active \xdef\@tmp{\scantokens{#1\noexpand}}\endgroup
      \expandafter\implicitgrpfork\@tmp\@tokcycle}% ACTIVE CHAR \bgroup
  }}
\long\def\@@grpT#1{\add@tcdepth\tctestifcon{\ifstripgrouping}{%
  \@grpT{#1}}{\groupedcytoks{\@grpT{#1}}}\sub@tcdepth\@tokcycle}
% \ COMMANDS (MACROS AND IMPLICITS)
\long\def\backslashcmds#1{%
  \test@ifmacro#1{\tctestifcon\ifcatSIX{\implicittoktrue\@chrT#1}{\@macT#1}}%
    {\implicittoktrue\trapimplicitegrp#1\implicitgrpfork#1}}
% FORK BASED ON IMPLICIT GROUP TREATMENT
\def\implicitgrpfork#1{\tctestifcon{\iftc@implicitgrp}{\ifcase
  \@implicitgroupingcase\or\addcytoks{#1}\or\@chrT{#1}\fi}{\@chrT#1}}
% SET UP ESCAPE MECHANISM
\def\settcEscapechar#1{\let\@tcEscapeptr#1%
  \def\tc@escapecytoks##1#1{\addcytoks{##1}\@tokcycle}}
\def\tc@trapescape#1{\tctestifx{\@tcEscapeptr#1}}
% TRAP CAT-6
\long\def\trapcatSIX#1{\tctestifcatnx#1\relax{}{\trapcatSIXb#1}}
\def\trapcatSIXb#1{\expandafter\tctestifcatnx\sv@hash#1{\catSIXtrue\trapcatSIXc#1}{}}
\def\trapcatSIXc#1{\tctestifnum{\count@stringtoks{#1}>1}{\tc@defx\six@str{\string#1}%
  \global\let\implicitsixtok\six@str\tc@tok{\implicitsixtok}}%
  {\tc@tok\expandafter{\string#1}\tctestifnum{\number\catcode`#1=6}%
  {}{\activetoktrue\implicittoktrue}}}
% DIRECTIVES FOR HANDLING GROUPED OUTPUT; DEFINE tokcycle GROUPING CHARS
\long\def\groupedcytoks#1{\begingroup\cytoks{}#1\exit@grouped}
\def\defineexit@grouped#1{\def\exit@grouped{\expandafter\endgroup\expandafter
  \addcytoks\expandafter{\expandafter#1}}}
\def\settcGrouping#1{\def\@tmp##1{#1}\tc@defx\@@tmp{\@tmp{\the\cytoks}}%
  \tc@earg\defineexit@grouped{\@@tmp}}
% FAUX TOKENIZATION OF COMMAND NAME (WHEN \ AND COMMAND-NAME TOKS ARE NOW cat12)
\def\csmk#1{\def\csaftermk{#1}\toks0{}\@csmkA}
```

```
\def\@csmkA{\futurelet\@tmp\@csmkB}
\def\@csmkB{\tctestifx{\@tmp\tc@sptoken}%
  {\toks0{ }\expandafter\@csmkF\tc@absorbSpace}{\@csmkCA}}
\def\@csmkCA#1{\tc@addtoks0{#1}\tctestifnum{\number\catcode`#1=11}%
  {\futurelet\@tmp\@csmkD}{\@csmkF}}
\def\@csmkC#1{\tctestifnum{\number\catcode`#1=11}
  {\tc@addtoks0{#1}\futurelet\@tmp\@csmkD}{\@csmkE#1}}
\def\@csmkD{\tctestifcatnx 0\@tmp\@csmkC\@csmkE}
\def\@csmkE{\tctestifx{\@tmp\tc@sptoken}%
  {\expandafter\@csmkF\tc@absorbSpace}{\@csmkF}}
\def\@csmkF{\tc@defx\thecs{\csname\the\toks0\endcsname}\csaftermk}
% TRAP IMPLICIT END GROUP TOK (e.g., \egroup); SET \iftc@implicitgrp
\def\trapimplicitegrp#1{\tctestifx{#1\egroup}{%
  \implicittoktrue\tc@implicitgrptrue}{}}
% TRAP ACTIVE TOK
\def\trapactives#1{\trapactivechar{#1}\trapactivetok{#1}}
\def\trapactivechar#1{\tctestifnum{\number\catcode`#1=13}{\activechartrue}{}}
\def\trapactivetok#1{\tctestifcatnx~#1{\activetoktrue}{\trapactivetokunexpandable#1}}
%% WILL ALSO TRAP ACTIVE \let TO PRIMITIVES AS IMPLICIT; UNDO LATER IN \can@absorb@@
\def\trapactivetokunexpandable#1{\tctestifcon{\expandafter\if
  \detokenize{#1}#1}{}{\activetoktrue\activetokunexpandabletrue\implicittoktrue}}
% FEATURES TO LOOK-AHEAD INTO THE INPUT STREAM (INTRODUCED v1.4)
\long\def\tcpeek#1#2\@tokcycle{\def\tc@tmp{\ifx#1\endtokcycraw
  \let#1=\empty\fi#2\@tokcycle}\futurelet#1\tc@tmp}%_____________________________PEEK_
\def\tcpopliteral#1{\tcpopwhitespace#1\tcpeek\@tmp\ifx\@tmp\bgroup
  \tcpop\@tmp\def\tc@tmp{\tcappto#1from}\expandafter\tc@tmp\expandafter
  {\expandafter{\@tmp}}\else\tcpopappto#1\fi}
\def\tcpop{\long\def\tc@@@pop##1{\tctestifx{\endtokcycraw##1}{\def@popname{}%
  \tc@tmp\endtokcycraw}{\def@popname{##1}\tc@tmp}}\tc@pop}
\long\def\tcpopuntil#1{\long\def\tc@@@pop##1#1{\def@popname{##1#1}\tc@tmp}\tc@pop}
\long\def\tc@pop#1#2\@tokcycle{\def\tc@popname{#1}\def\tc@tmp
    {#2\@tokcycle}\futurelet\tc@futuretok\tc@@pop}
\def\tc@@pop{\tc@trapescape\tc@futuretok{\def@popname{}\tc@tmp}{\tctestifx{%
  \endtokcycraw\tc@futuretok}{\def@popname{}\tc@tmp}{\tctestifx{\tc@sptoken%
  \tc@futuretok}{\spacepoppedtrue\tc@@@pop}{\spacepoppedfalse\tc@@@pop}}}}%_____POP_
\def\tcappto#1from#2{%
  \expandafter\tc@defx\expandafter#1\expandafter{\expandafter#1#2}}
\def\tcpopliteralappto#1{\tcpopliteral\@tmp\tcappto#1from\@tmp}
\def\tcpopappto#1{\tcpop\@tmp\tcappto#1from\@tmp}%_____________________________APPEND_
\long\def\tcpopwhitespace#1#2\@tokcycle{\def\tc@popname{#1}\def@popname{}%
  \def\tc@tmp{#2\@tokcycle}\futurelet\tc@futuretok\tc@popspc}
\def\tc@popspc{\tctestifx{\tc@sptoken\tc@futuretok}{\discern@space}{\tc@tmp}}
\def\discern@space{\begingroup\def\@@spcT##1{\tctestifcon\ifimplicittok{\gdef
  \tc@nxt{\tc@tmp##1}}{\gdef\tc@nxt{\def@popname{##1}\tc@tmp}}\endgroup
  \tc@nxt}\stringify\@@@@@@spcT}%_____________________________________________SPACE_
\long\def\@tcpush#1#2\@tokcycle{\def\tc@tmp{#2\@tokcycle}\expandafter\tc@tmp#1}
\long\def\@tcpushgroup#1#2\@tokcycle{\def\tc@tmp{#2\@tokcycle}\expandafter\tc@tmp
  \expandafter{#1}}
% ...BORROW \addcytoks OPTIONAL ARGUMENT EXPANSION FEATURE FOR \tcpush[group]
\def\tcpush{\bgroup\long\def\tc@addtotoks##1{\egroup
  \@tcpush{##1}}\futurelet\nxttok\addcytoks@A}
\def\tcpushgroup{\bgroup\long\def\tc@addtotoks##1{\egroup
  \@tcpushgroup{##1}}\futurelet\nxttok\addcytoks@A}%_____________________________PUSH_
% EXPRESS-INTERFACE - ALLOWS TO EXTERNALLY DEFINE DIRECTIVES
\def\Characterdirective{\def\@chrT##1}
\def\Groupdirective{\long\def\@grpT##1}
\def\Macrodirective{\long\def\@macT##1}
```

```
\def\Spacedirective{\def\@spcT##1}
% EXPRESS-INTERFACE - DEFAULT DIRECTIVES
\def\resetCharacterdirective{\Characterdirective{\addcytoks{##1}}}
\def\resetGroupdirective{\Groupdirective{\processtoks{##1}}}
\def\resetMacrodirective{\Macrodirective{\addcytoks{##1}}}
\def\resetSpacedirective{\Spacedirective{\addcytoks{##1}}}
\def\resettokcycle{\resetCharacterdirective\resetGroupdirective
  \resetMacrodirective\resetSpacedirective\aftertokcycle{}%
  \stripgroupingfalse\stripimplicitgroupingcase{0}}
% SUPPORT MACROS FOR TOKENIZED OUTPUT: \addcytoks[<expansion level>]{<arg>}
% (CONTRIBUTED BY CHRISTIAN TELLECHEA)
\def\addcytoks{\futurelet\nxttok\addcytoks@A}
\long\def\tc@addtotoks#1{\cytoks\expandafter{\the\cytoks#1}}
\def\addcytoks@A{\tctestifx{[\nxttok}\addcytoks@B\tc@addtotoks}
\long\def\addcytoks@B[#1]#2{\tc@ifempty{#1}\tc@addtotoks
  {\tctestifx{x#1}{\tc@xarg\tc@addtotoks}{\addcytoks@C{#1}}}{#2}}
\def\addcytoks@C#1{\tctestifnum{#1>0}{\tc@earg\addcytoks@C
  {\the\numexpr#1-1\expandafter}\expandafter}\tc@addtotoks}
% SET INITIAL PARAMETERS
\settcGrouping{{#1}}%          E.G. <<#1>> IF cat-1,2 SET TO < AND >
\settcEscapechar{|}%           BYPASS TOKCYCLE PROCESSING BETWEEN |...|
\resettokcycle%                WHICH ALSO CONTAINS THE FOLLOWING 3 RESETS:
%  \stripimplicitgroupingcase{0}% DEFAULT RETAIN UNALTERED \b/e-groups
%  \stripgroupingfalse%           DEFAULT RETAIN UNALTERED {} GROUPING
%  \aftertokcycle{}% NO DEFAULT CODE EXECUTED AFTER EACH TOKCYCLE INVOCATION
\restorecatcode
\endinput

EDIT HISTORY
v1.0  2019/08/21
      - Initial release

v1.1  2019/09/27
      - Introduced \ifactivechar, \ifactivetokunexpandable
      - Tightened up consistent definition of implicit (to exclude primitives)
      - Rewrote active token trapping logic, to differentiate between active
        token vs. active character code, in the event that an earlier tokenized
        token no longer shares the current characteristics of the character code
      - Added ability to handle active-implicit grouping tokens
      - Added ability to handle active-implicit cat-6 tokens

v1.11 2020/02/04
      - Fixed bug in \can@absorb@@ macro, which prevented the proper absorption/
        handling of the = token.

v1.12 2020/02/11
      - Documentation correction: \tokcycleenvironment, not \tokencycleenvironment
      - Documentation correction: misspelling in tokcycle-examples.tex
      - Redefined \tc@defx and \tc@earg to omit #2 as part of definition
      - Corrected \trapcatSIXb definition to account for revised \tc@earg definition.

v1.2  2020/10/01
      - Added/fixed capability to handle active-implicit spaces. While the
        #1 passed to the \Spacedirective, in such a case, is an implicit
        space \tc@sptoken, the name of the active character from whence it
        originated in the input stream is stored as an explicit cat-12 token
        in the definition \theactivespace.  The flags \implicittok and \activetok
```

```
        are both set true, and the \activechar flag is checked, as well.

v1.3  2021/03/10
      - Introduced \xtokcycleenvironment, similar to \tokcycleenvironment, but
        allows two additional arguments, defining the setup and trailing code
        that will be run prior to the invocation and following conclusion of the
        token cycle.
      - Introduced \truncategroup, a directive to discard remaining tokens in the
        current token-cycle group and close the group.
      - Introduced \truncatecycle, a directive to discard remaining tokens in the
        token-cycle input stream, but closing any open groups.
      - Introduced \truncategroupiftokis{}{} to conditionally issue a \truncategroup
        if the current token under consideration matches the 1st argument.
      - Introduced \truncatecycleiftokis{}{} to conditionally issue a \truncatecycle
        if the current token under consideration matches the 1st argument.
      - Introduced \tcendgroup as a form of \endgroup that saves contents of
        \cytoks upon group exit.  Gives clarity to definitions of \tokencycle
        and \tokencyclexpress environments.  Available for general used.
      - Fixed bug.  Made \@@grpT \long, in the event that \par occurs inside a
        group.
      - Fixed bug.  Added \relax to end of \add@tcdepth definition, the absence
        of which had prevented timely update of the \tcdepth count.
      - Documentation correction: improved explanation of implicit-cat-6 token
        limitations.  Likewise, added warning regarding active-implicit spaces.
      - Documentation prepared with lmodern font, rather than default cm,
        for reasons of better PDF hinting.
      - Introduced \exit@grouped to the \groupedcytoks definition, for clarity and
        ease of \expandafter.
      - Reworked logic so that \tcdepth is associated with nested calls to the
        Group directive, rather than with each invocation of \processtoks.
        This primarily required redefinition of \@@grpT and its components.
      - Changed name of \tc@depth to \tcdepth, as it may be a useful parameter for
        users to check the token-cycle nesting depth.  Therefore, also renamed
        \subtc@depth to \sub@tcdepth, \addtc@depth to \add@tcdepth.
      - Excised \sub@tcdepth from \endtokcycraw as part of new logic making
        \tcdepth associated with the Group directive (also, outer token cycle
        operates with depth = 0 rather than 1).
      - Renamed \@defgroupedcytoks to \defineexit@grouped, to better match function.

v1.4  2021/05/26
      - Concerning the \@@@@spcT macro: previously, active-implicit spaces
        were passed to the Spacedirective as \tc@sptoken, with the \string of the
        active char passed in \theactivespace. Now, the active-implicit space token
        itself is passed to the Spacedirective instead of \tc@sptoken, but only *IF*
        the charcode of that character is currently active; otherwise, a generic
        implicit space, \tcsptoken is passed.
      - Introduced a set of "look ahead" macros: \tcpeek, \tcpop, \tcpopliteral,
        \tcpopappto, \tcpopliteralappto, \tcappto#1from#2, \tcpopuntil,
        \tcpopwhitespace, \tcpush, and \tcpushgroup.
      - Bug fixed in \tc@addtoks definition, in the event that #2 was a number.
```